

## Re: Volatile variables

**Source:** [http://coding.derkeiler.com/Archive/C\\_CPP/comp.lang.c/2004-02/5040.html](http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2004-02/5040.html)

---

**From:** srinivas reddy ([srinivasreddy\\_m\\_at\\_yahoo.com](mailto:srinivasreddy_m_at_yahoo.com))

**Date:** 02/28/04

Date: 27 Feb 2004 18:16:45 -0800

I think I didn't phrase my question properly. My program needs to have a volatile variable to work correctly and there are no bugs in the code. Now even if a variable declared as volatile, could program fail to read the modified value of volatile variable? Would compiler store volatile variable in cache? If so, then two processes cache same volatile variable at two different locations. One process can't see other process's modification unless variable is flushed from cache to memory (cache is implemented using write-back policy, so memory doesn't have updated value)?

Dan.Pop@cern.ch (Dan Pop) wrote in message news:<c1nmej\$pbf\$15@sunnews.cern.ch>...

> In <ff8ef364.0402270008.36b416b0@posting.google.com> [srinivasreddy\\_m\\_at\\_yahoo.com](mailto:srinivasreddy_m_at_yahoo.com) (srinivas reddy) writes:

>

> >Is there any chance that a program doesn't work properly even after a  
> >variable is declared as volatile?

>

> If a program was correct before declaring anything as volatile, it will  
> keep being correct after and will produce the same output, unless the  
> output depends on unspecified behaviour. The only purpose of volatile is  
> to dumb down the compiler, WRT certain optimisations otherwise allowed by  
> the language.

>

> OTOH, there are programs that aren't correct in the absence of the  
> volatile qualifier and that are fixed this way. Here's an example:

>

> #include <stdio.h>

> #include <signal.h>

>

> sig\_atomic\_t gotsig = 0;

>

> void handler(int signo)

> {

> gotsig = signo;

> }

>

> int main()

> {

comp.lang.c: Re: Volatile variables

```
> signal(SIGINT, handler);
> puts("Press the interrupt key to exit.");
> while (gotsig == 0);
> printf("The program received signal %d.\n", (int)gotsig);
> return 0;
> }
>
> Because gotsig is not volatile, the compiler is free to "optimise" the
>
> while (gotsig == 0);
>
> loop to:
>
> if (gotsig == 0) while (1);
>
> since it sees no way the value of gotsig can change inside the loop.
>
> If gotsig is volatile, the compiler must assume that its value can change
> behind its back and keep testing its value.
>
> But this doesn't mean that it's worth trying to fix broken programs by
> randomly throwing in volatile qualifiers. As usual, there is no
> substitute for knowing what you're doing.
>
> >I remember somebody mentioning a
> >scenario involving L1, L2 caches. Could anybody throw some light on
> >this?
>
> Whoever mentioned such a scenario was heavily confused and in dire need
> of a clue. The abstract C machine has no caching, therefore caching is
> irrelevant to the correct behaviour of a C program.
>
> Another typical example of using volatile is when writing memory testing
> programs. Such programs often write values in memory and then read them
> back and compare with the original. To the compiler, it is obvious what
> the result of the comparison should be, so it can optimise all the memory
> testing away and declare that the memory works correctly. To force the
> writing to and the reading from memory, you *have* to use a pointer to
> volatile data.
>
> Dan
```